BT011

How Regular Expressions Really Work

Jack N Shoemaker, Greensboro, NC

ABSTRACT

Regular expressions are a powerful and ubiquitous tool for manipulating text and data employed by many Unix editors and utilities like sed, awk, perl, vi, and emacs. This paper is divided into two parts. So the reader can begin to understand how regular expressions work in general, the first part will describe the basic parts of a regular expression: character classes, meta-characters, quantifiers, grouping, and anchoring. The second part will focus on the specific SAS® implementation using a real-life example culled from a SAS-L discussion list thread. </P>

INTRODUCTION

This paper is not intended to be a comprehensive overview and tutorial about regular expressions. For that the reader is directed to the book referenced at the end of this paper in the bibliography section. Rather, the aim of this paper is to provide the SAS programmer with a working knowledge about the structure, syntax, and lexicon of regular expressions so that the SAS RX functions may be used with confidence and efficiency.

Regular expressions provide a mechanism for describing character-string patterns. A host language, like SAS, uses these patterns to match and perhaps change or delete the character string found during the pattern search. Consider a common PC task – locating all the SAS programs in a particular folder. From the command prompt you would type something like 'dir *.sas'. Note the use of the asterisk. You have employed the wild-card character that means match any file name. The '.sas' part of the 'dir' command is taken literally to mean files with an extension of 'sas'. So, the argument to the dir command is made up of two pieces – a meta-character that has special meaning and a string literal that is interpreted literally. Though much more powerful than DOS wild-card characters, regular expressions are essentially the same thing – a form of syntax that allows you to express a variety of string patterns in a short and compact form. In this respect, regular expressions are a mini programming language devoted exclusively to the task of pattern matching.

WHAT'S SO REGULAR ABOUT REGEX?

The term regular expression comes from models described in formal algebra as regular sets. If you are mathematically inclined and would like to know more about the mathematics behind these theories do a Google search on Stephen Kleene. Regular expressions (or regex, for short) remained primarily in theoretical circles until 1968 when Ken Thompson published the first computational algorithms for regular expressions. The point is to not get too bogged down with the technical meaning of regular. Just think of regex as mini programming language contained within a larger host language like Perl, Python, or SAS.

REAL-LIFE EXAMPLE

Consider this real-life example posted on SAS-L a while back. A medical researcher had notes and observations about sick kids. The researcher wished to identify all children who had a sore throat. In the relevant column that contained this information, a sore throat could appear as 'ST', 'S.T.', 'sore throat', 'Sore Throat', 'ST.', or a variety of other permutations on this theme. A number of clever solutions were proposed by the SAS-L community involving compressing out the periods and spaces, and using the indexw() and scan() functions to locate and extract the desired tokens. The researcher finally settle on a regex solution using the rxmatch() and rxparse() functions instead. Keep this example in mind as we take a quick tour of regex syntax.

THE PARTS OF A REGULAR EXPRESSION

There are five basic parts to regular expressions that you need to understand in order to make regular expressions work for you. Those parts are:

- meta-characters – characters which have special meaning in the pattern akin to the DOS wild-care characters for filename pattern matching
- character classes – sets of characters any one of which will match the pattern
- quantifiers – a meta-character, that indicates how many times a particular sub-pattern will repeat
- groupers – a meta-character used in conjunction with quantifiers to delineate the pattern to repeat
- anchors – a meta-character specifying a line or word boundary

A careful reader might notice that everything appears to be just a meta-character or a character class. True though this is, quantifiers, groupers, and anchors are different enough to deserve a special mention on their own.

Each of these basic parts of the regular expression will be described below. Bear in mind that each host language has its own dialect of regular expression. For the purposes of this paper we will focus on those elements thought to be almost universal. Notwithstanding, before using a regular expression you should always consult your host documentation for any nuances or improvements on the basic syntax.

META CHARACTERS

The meta character to match any one character is a dot '.'. So the regular expression 's.s' will match 'sas', 'sos', or any other three-letter string that starts and ends with an 's'.

The meta character to match one pattern or the other is the pipe character. So the expression

```
'this|that'
```

will match either the string 'this' or the string 'that'.

The meta characters used to group patterns into a single unit are the parentheses. For example, the expression

```
'I want (this|that)'
```

will match 'I want this' or 'I want that'.

CHARACTER CLASSES

Character classes are similar to the dot meta character except that the character class specifies a range of possible matches.

That is the pattern will match any one of the characters listed in the character class. The list of characters is placed inside square brackets. For example, in the example above, the indication of a sore throat may appear as 'st' or 'ST' among various other constructions. To write a regular expression to match either 'st' or 'ST' we would write

```
'[Ss][Tt]'
```

This is actually two character classes – '[Ss]' and '[Tt]'. This regular expression will match any two-character string that starts with 'S' or 's' and ends with 'T' or 't'. Note that this expression will also match 'sT' and 'St'. That may be an added side benefit or cause for additional specification depending on the details of your application.

Now consider a related task. You have a list of SAS variable names to check. As you know, SAS variable names may not start with a number. In regular expression thinking you want a negated character class. That is, a character class that matches any character not listed in the character class. The syntax for negating a character class is to place a caret character ('^') as the first character which the character class. To write a regular expression to match any character string that does not start with a number, you would write

```
'[^0123456789].*'
```

Note that this regular expression contains a quantifier ('*') that we will discuss in the next section. More importantly, this regular expression is not sufficient to check for valid SAS variable names because it would allow the first character to be '!', '@', '#', or any other non digit for that matter. The point is that a character class is negated if the first character is a caret. If a caret appears anywhere else in the string, then it is treated as a normal member of the character class list. The reader may wish to consider what this regular expression would mean

```
'[^^]'
```

The answer is that this will match any character except a caret. Matching any single digit is a fairly common task. You might wonder if there is some short hand for specifying any number, or for that matter any letter. The answer is 'yes' with the following caveat. The area of character-class notation is one where syntax varies wildly across host languages. SAS for example offers a dizzying array of short hands for character classes. You may find these handy and convenient; however, give some consideration to portability to other host languages before becoming overly enamored with the SAS character classes. In any event here are some short hands that are bound to work in almost any host language that supports regular expressions.

- Any digit – `'[0-9]'`
- Any letter – `'[a-z]'`
- Any capital letter – `'[A-Z]'`

The negation character listed above is a character-class meta character. Note in the bulleted list above we have introduced another character-class meta character, the dash ('-'). Inside a character class, the dash indicates a range of characters. What happens if the first character in a character class is a dash? Well it can't possibly indicate a range, so it is just treated as a member of the character class.

So character classes have their own set of meta characters. And the rules governing these meta characters are completely different inside and outside the character class. Let's consider another example to drive this point home. We would like to match the date of this paper's presentation at SUGI 28 – March 31, 2003. For the sake of this example let's assume that the date is expressed as '03/31/2003', '03-31-2003', or '03.31.2003'. That is the month, day, and year are separated by a forward slash, dash, or dot. From the discussion of the dot meta character above you might be tempted to use this simple regular expression.

```
'03.31.2003'
```

This will certainly match the three desired date formats above; however since the dot meta character matches any character it will also match strings like '03x3182003'. We can improve the regular expression by considering the desired pattern as three character literals – '03', '31', and '2003' connected by a single character in the class '-./'. For example,

```
'03[-./]31[-./]2003'
```

Note that the dash character is the first character in the character class. If it were not the first character, it would have been treated as the character-class meta character indicating a range of characters. This is not what we would have wanted in this situation. Next consider the dot character. Isn't this the meta character meaning 'match any single character'. Yes it is if outside a character class. Inside a character class, the dot character is just like any other character.

At first the regular expression listed above may seem a bit odd and impenetrable. However, once you sit back and break the regular expression into its components, the meaning becomes clearer. Drawing on the mini programming language analogy, this is similar to reviewing a new program for the first time. The intent and flow of the program becomes clearer once you break it down into discrete parts.

QUANTIFIERS

Our discussion of negated character classes introduced the asterisk character as a meta character know as a quantifier. A quantifier specifies how many times the immediately preceding character or sub-expression should be repeated. There are only three quantifiers

- ? match zero or one
- * match zero or more
- + match one or more

That's it. So the expression from above `'.*'` means match zero or more of any character. Consider the task of matching a time stored in 24-hour format. That is, HH:MM, where HH, may or may not have a leading zero.

Let's attack the minutes portion first since that's easier and will not require quantifiers. From our previous discussion on character classes and short hands, we can view the minutes portion of the time as a two-character string which begins with the digit 0 through 5 followed by any digit 0 through 9. We can form the corresponding regular expression by translating the sentence above into regex.

```
'[0-5]'[0-9]'
```

The hours portion of the time requires a bit more thought. One way of describing the hours portion is an optional 0 followed by a 0 through 9 or a 1 followed by a 0 through 9 or a 2 followed by a 0 through 3. That's three alternatives. The first alternative involves an optional leading 0 character. Using the ? quantifier we would write this as

```
'0?[0-9]'
```

That is, zero or more 0 characters followed by a single character from the class of characters 0 through 9. The second alternative is

```
'1[0-9]'
```

That is, a 1 character followed by a single character from the class of characters 0 through 9.

The final alternative is

```
'2[0-3]'
```

That is, a 2 character followed by a single character from the class of characters 0 through 3.

So putting these three parts together we would have

```
'0?[0-9]|1[0-9]|2[0-3]'
```

Note that we can describe the first two alternatives as a single alternative. Namely, an optional 0 or 1 character followed by a single character from the class of characters 0 through 9. Translating this sentence into to regex we have

```
'[01]?[0-9]
```

So we can write the entire expression as just two alternatives as follows:

```
'[01]?[0-9]|2[0-3]'
```

There is another way to view this problem as a choice between two alternatives. An optional 0 or 1 character followed by a single character from the class 4 through 9, or an optional 0, 1, or 2 character followed by a single character from the class 0 to 3. Translating to regex we have

```
'[01]?[4-9]|[012]?[0-3]'
```

In all these cases we have used the question-mark quantifier to specify zero or one occurrence of the character immediately preceding the quantifier. Note as well the process we have used to create the regular expressions. First, we write out the pattern in standard English, then we translate the sentence into regex. Of course it helps to be in a regex frame-of-mind when writing out these sentences.

GROUPERS

We have seen an example of the grouping meta characters previously in the 'I want this' or 'I want that' example. We can use the grouping meta characters to complete our example from above. We now know how to construct a regular expression for the hours portion of the time and the minutes portion of the time. All that is left is to bring these two sub-expressions together and join them with a colon.

```
'([01]?[0-9]|2[0-3]):[0-5][0-9]'
```

The parentheses are used to group the hours sub-expression. The colon is just a character literal in the regular expression string. Finally the two character classes [0-5] and [0-9] complete the specification for the time format. Staring at the expression above without context might make you think you had encountered some memory overflow error that had caused your computer to begin echoing all manner of garbage to the screen. Once you knew that this expression was a regular expression you could begin the task of breaking it down into its components and begin to understand what was intended.

ANCHORS

Most implementations of regular expressions work with lines of text. So, there is a natural need to specify the beginning of a line and the end of a line. Since these concepts are really cursor positions and not actual characters, there are two meta characters used to indicate each position.

- Beginning of line - ^
- End of line - $

We discussed the meaning of the caret character previously in the discussion about negated character classes. We concluded that discussion with the statement that the position of a meta character inside or outside of a character classes changes the meaning of the meta character. It is worth repeating that a caret outside a character class which is the first character in a regular expression means 'match the start of a line', while if the caret appears elsewhere it means just match a caret character.

Since most SAS applications will operate on fields or columns and not lines, we won't devote any more attention to these meta characters. In addition to line boundaries, many modern version of regular-expression processors contain notation for the beginning and ending of words. Not surprisingly, SAS has a number of these; however, since this first section was meant to be a universal introduction to regular expression syntax, we won't go into those details at the moment.

With this brief introduction under our belts, let's turn our attention to a real-life problem and see how we can use the SAS implementation of regular expressions to solve it.

WHO HAS A SORE THROAT?

A researcher at Children's Hospital in Boston wanted to locate all records indicating a sore throat. Here is an excerpt from her SAS-L posting

```
I'm trying to pull out all records where a
character variable has the value ST.  This is an
abbreviation for sore throat and here are some
examples of how it might appear.
      256    N/A EARACHE,ST
166084    ST
166099    ST URI SX
161846    ST, LARYNGITIS
161848    ST,PROD COUGH
160700    FEVER,  ST
161973    FEVER ST
135329    VIRAL SX ST
```

Furthermore, she did not want to locate false positives like these

```
2927    CHEST PAIN
2950    L BREAST LUMP
2964    NEEDLE STICK
2988    WRIST PAIN
3000    CAST EVAL
```

In other words, the task is more difficult than just searching for the string 'ST' anywhere on the line. She did not want to locate occurrences of 'ST' when 'ST' was part of some other word like chest, breast, stick, wrist, or cast.

The first set of responses suggested using some combination of the translate() indexw() functions to search for blank-delimited words of the form 'ST'. Here is a typical suggestion from that category.

```
y = indexw( tranwrd( word, ',' , ' '), 'ST');
```

In other words, change commas to blanks using tranwd() and then use indexw() to locate the 'ST' tokens. The resulting value of y would hold the starting position of 'ST' in the original WORD variable. If y were non-zero, you had a match.

This solution wouldn't work if some other piece of punctuation needed to be considered, like '/' or '.', but is easily changed to handle these situations.

USING SAS RX FUNCTIONS

The researcher wanted a regular-expression solution. To use regular expressions in SAS, there are primarily two functions to keep in mind:

Rxparse() – converst a regular expression into some sort of internal SAS representation
Rxmatch() which uses the result of rxparse() and a target string to find a match

Although a discussion of the SAS data step and how it handles variable initialization is beyond the scope of this paper, suffice it to say that on each loop of the data step, all data-step variables are set to missing and then assigned values as part of the subsequent program logic. It is typical, that you will use the same regular expression for the entire duration of the data step, so it saves time to just call rxparse() once to parse the regular expression and turn it into its internal representation. To instruct SAS not to reset this result to missing with each loop of the data step, you use the RETAIN statement. So a typical data-step structure using regular expressions looks something like this:

```
data abc;
  set def;
  /* retain value of rx */
  retain rx;
  /* assign value on first loop */
  if _N_ = 1 then
   rx = rxparse( /* regular expression */ );
  …
  /* use rxmatch() to find sub-string */
  hit = rxmatch( rx, string-to-search );
  run;
```

Here was the researcher's first attempt at solving the problem using regular expressions.

```
rx = rxparse("' ST'|'/ST'|',ST'|'ST '|
  'ST\'|'ST/'|'ST,'|'.ST'|'S.T.'");
pos = rxmatch( rx, complaint );
```

What we have is a long list of alternatives – nine in all – being checked in a data-step variable called 'complaint'. This was little better than just searching for the string 'ST' since CHEST, WRIST, etc. would still be matches using the above.

The next iteration was to use negated character classes to make sure that 'ST' had clear space on both sides. The following was suggested by one of the SAS-L respondents.

```
rx=rxparse("^'0-9A-Za-z' ST ^'0-9A-Za-z'");
```

The idea was to not have 'ST' appear with a number or letter before it or after it. Note the syntax of this regular expression. Character classes are enclosed within single quotes; the negation operator happens outside these single quotes – much like any SAS negation operator; and, the whole thing appears within double quotes.

This solution still doesn't find patterns like 'S.T.' or 'S/T'.

Let's write a sentence describing what we want to find and then translate that sentence into SAS regex. We want to find words that start with 'S', end with 'T' and are either only two characters long, or have only a comma or period in between the 'S' and 'T'.

In the more generalized discussion of regular expressions, I noted that in addition to line-begin and line-end markers, many flavors of regular-expression processors contain anchors for word boundaries. SAS is no exception. The prefix '$p' instructs SAS to look for words that start with the following character. The suffix '$s' instructs SAS to look for words that end with the following character. So the expression

```
"$p S T $s"
```

will match any two-character word which begins with an 'S' and ends with a 'T'. In order to account for the embedded punctuation, we can supply an optional character class

```
"[$'.,']
```

in between the S and the T in the previous expression. The final regular expression pattern that the researcher felt solved her problem looked like this

```
"$p S [$'.,'] T $s"
```

CONCLUSION

The good news is that SAS has a regular expression processor and that it may be used to solve a variety of pattern-matching problems that may not be as easy to do or understand otherwise. Using a process of writing down the problem in English and then translating to a regular expression often leads to the best solution. The bad news is that the SAS regular expression syntax is different enough from standard regular expression syntax that it may take users of other regular-expression processors a moment to get used to. Version 9 of the SAS system has a separate set of "P" regular expression functions which pass the regular-expression task out to an external Perl processor. This will certainly make the syntax more in line with other utilities – like Perl; however, the ultimate performance penalty, if any, remains to be tested.

REFERENCES

Friedl, Jeffery E.F., *Mastering Regular Expressions*. Sebastopol, CA: O'Reilly & Associates, Inc., 1997.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Jack N Shoemaker
Greensboro, NC
shoe@theworld.com